**CSC220 Assignment06**

**Sorting and Timing**

The goal of this week’s assignment is:

1. Practice sorting
2. Practice measuring the runtime of a program
3. Practice working with generics
4. Continue learning the significance of special cases!
5. Learning how to write test to check your implementation!

**Things you must do:**

1. There are many details in this assignment. Make sure you read the whole thing carefully before writing any code and closely follow this instruction.

2. You must complete your assignment **individually**.

3. Always remember Java is case sensitive.

4. Your file names, class names, and package name must match exactly as they are specified here.

**Things you must not do:**

1. You must not change the file names, class names, package names.

2. You must not change the signature of any of these methods (name, parameters, …).

For this assignment, you are asked to construct a program that has the capability to determine if two words are anagrams and to find the largest group of anagrams in a list of words. Two words are anagrams if they contain the same letters in the same frequency. For example, alert and later are anagrams.

Now that we have some basic functionality in AnagramUtil and a working insertion sort implementation, we are ready to deal with the problem of the largest group of anagrams in a list of words. We will also turn to a more efficient sorting algorithm than the one we saw in lab, i.e. merge sort, and predict its runtime for very large input sizes.

**Part 0**

• You first must make sure that you have already finished the lab successfully and have all methods in the lab instruction working properly.

• **DO NOT** start your assignment unless you have all the features in the labworking.

**Part 1 –** AnagramUtil **implementation: phase 2**

To find the largest group of anagrams in a list of words, sort the list with a Comparable that compares the sorted character representations of the words. After the sort, any group of words that are anagrams of each other will be adjacent in the list. We will concern ourselves only with word anagrams and not anagrams that are phrases, which may contain whitespace and punctuation.

* public static String[] getLargestAnagramGroup(SortedString[] stringList)
  + This method returns the largest group of anagrams in the input array of SortedString words, in no particular order. It returns an empty array if there are no anagrams in the input array.
  + The starting code contains lots of hints on how to implement this. You need only to fill the remaining parts. Please follow this algorithm.
  + You may assume that the word list given as input does not contain duplicates.

Before moving to Part 2, **test** your getLargestAnagramGroup implementation in Tester.java. To make your job easier, we are providing a sample set of words for you called: **sample\_word\_list.txt**. You can load it into Eclipse the usual way; if you have doubts, there are instructions at the end of this document.

* Once loaded, the following line can be invoked to test your implementation (note how this first calls an overloaded method which then calls the method *you* wrote). You should verify the returned array s3 is correct:   
  String[] s3 = getLargestAnagramGroup("../sample\_word\_list.txt");
* The output returned should be: carets, Caters, caster, crates, Reacts, recast, traces.

Here is also a list of more scenarios you should consider. It would be easiest to modify sample\_word\_list.txt or create your own text file for these test cases:

* an empty file
* a file with one word
* a file with two words
* a file with no anagrams
* a file that has the largest anagram group in the middle of the file
* a file that has the largest anagram group at the end

**Part 2 -** MergeSort **implementation**

We will now implement our second sorting algorithm using a merge sort.

* public E[] sort(E[] array)
* This method takes a generic input array and returns a sorted array using a merge sort. Since this method is generic, we cannot use > or < for comparing values. Instead, we must make use of Comparable, a la compareTo().
* Most of the work here has already been completed for you. The remaining work is to complete the private sort(), which the public sort() calls. Look for the comments to see what needs to be finished.
* The end of this document contains an example of merge sort if you are in doubt. Also consult your lecture notes.
* Before moving to Part 3, **test** your MergeSort implementation in Tester.java. You should check the following with MergeSort<Integer> :
  + a list with one element
  + a list with two elements
  + a sorted list of numbers
  + a random list of numbers
* As a final check, try using MergeSort<SortedString> instead of InsertionSort<SortedString> inside getLargestAnagramGroup(). The output should not change.

**Part 3 - Predicting runtime for** MergeSort

For the last part of the assignment, we will measure the performance of MergeSort using the time formula you saw in lab. This formula is quite useful as it allows us to make *predictions* about how our function will do for much larger inputs, without actually running it.

* public double O(int n)
  + This method returns the order O() of the implementation.
  + Consult the lecture slides if you have doubts
  + Note: for a runtime of O(n2), it is better to use Math.pow(n, 2) rather than n \* n.
* public void fit(E[] array)
  + This method calculates the constant c using the given input array. Time measurements are measured in microseconds.
  + Remember the formula has form time = c \* O(), where time is the time it takes to run the insertion sort for the input array and O() is the big-Oh given from the function above.
  + To test the merge sort algorithm, make a call to sort().
  + To time your program, you can use System.nanoTime(). Remember to convert nanoseconds to microseconds!
* public double predict(int n)
  + This method predicts the running time of an insertion sort for an input size n. The estimated time return is in unit microseconds.
  + Note: this method will **NOT** run the insertion sort with the given n. Instead, you **must** use the calculated constant c from fit() to make a prediction for the n. You may assume that fit() has been called previously before predict().

To test your timing functions, try running fit() with a modestly sized array, say with 10 elements. Then try running predict() for an input size much larger, say 100K or even 1M. How much time would it take to sort such a large list? How does this compare with your values for insertion sort? Think about whether your answers make sense given what you know about merge sort. If you have doubts, consult your instructor, the TAs, and LAs.

**Part 4 - Testing**

Unlike previous labs/assignments you are not given any tests as a starting point. You must create your own tests to examine all and every method you implemented in the previous part. Several testing suggestions were made throughout the lab. Make sure you check the functionality of your methods very carefully.

**Remarks**

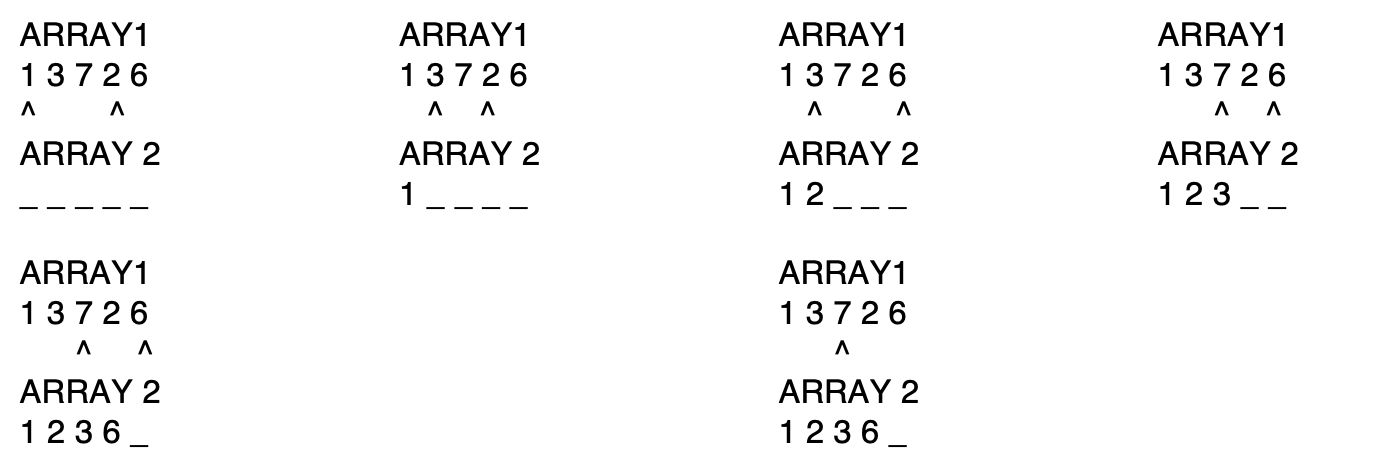
* Make sure to submit your assignment by (re-)uploading your **Lab06** folder into your **csc220-cXXXX** folder by the deadline **(Tuesday @ 11:59pm)**
* Be sure to include **Tester.java** in your submission to BOX. Remember that we will **not** run your testing code; we will only confirm that you have done testing.
* **For all your assignments, please start early and seek help early (either from the instructor or the TAs).**

**Merge Sort Example**

Here is a worked merge sort example. Consider the following array of integers:

7 3 1 6 2

Through recursion this becomes two sorted pieces:   
  
1 3 7 (from index 0 to 2)   
2 6 (from index 3 to 4)   
  
Remember that these are part of the same array, array1. We then take these two arrays and add them to a result array called array2.



It is important to note that in the last step, the b pointer “ran” out. At that point, we can just add everything over from the other pointer, which is a.

1 2 3 6 7

**Loading the sample\_word\_list.txt**

• Right click on Lab06 (i.e., the project name – not the src folder) and then “new->file” and type “sample\_word\_list.txt”. A file with the same name will show up under “JRE System Library”.

• Grab a copy of sample\_word\_list.txt from the Lab06-Assignment06 ZIP folder from the Google Drive and paste the content of it into the file you just created.

• You can now do (in Tester.java):   
String[] s3 = *getLargestAnagramGroup*(“sample\_word\_list.txt”);

• should return an array s3 that contains the following strings, in no particular order: carets, Caters, caster, crates, Reacts, recast, traces.

• if you are seeing anything else, go back and debug your code.

Again, this is only one check to double check your code. After you got the expected

output from sample\_word\_list.txt, you need to think of other tests. Think about what

would happen if the array of words is empty or has no anagrams. Is your function handling those cases correctly? If not, go back and debug.